(Edited Feb. 22, 2022)

Introduction

Design the controller of your microprocessor.

Controller

The first step in designing the controller is to identify and define the control signals, which should already be documented in CAD7. The controller for the baseline architecture is simplified by the fact that all instructions are one word long. In the simplest implementation of the microcontroller, the instruction decode can be done without finite state machines (FSMs). If FSMs are used for jump and branch, or other instructions (e.g. interrupt processing) which require two cycles in execute stage, the FSMs can be very simple.

Implementing the control in a pipelined machine which has a separate decode stage (unlike ours) means having the necessary logic to set the control bits to their respective values in each stage for each instruction. This is accomplished by decoding the opcode from the IR and the condition bits (in case of a branch of jump) to create the control bits which are stored in the Control Register. These control bits dictate the function of the EXECUTE stage in the next cycle. Simple control is also required for the fetch stage; the details of which depend upon your timing scheme for memory access and write back.

In the 427 Baseline Architecture, decode and execute are in the same stage (to avoid data dependencies). The decoder is so simple and fast that it does not add much delay to the execute stage. You can implement the control (decode) logic without a control register but be certain that any lines controlling writing to data memory or register files are hazard-free.

Your data memory interface (and sometimes your program memory interface) will have associated control signals in addition to data and address busses. These control signals should also be driven by your control unit. You may also need a few flip-flops to save certain control bits for one more clock cycle; for example, if you write back results into the register file on the positive edge of the clock after the completion of the execute stage, you may need to latch the Write_Enable signal on the falling edge of the clock so that you do not use the control signal value of the next instruction. Consider making some or all of the flip-flops in your controller scannable. If you don't have an external program memory interface, scanning the IR will be important. Scanning the PSR is also a good idea. Base your decision to scan other flip-flops, if present, by considering their controllability and observability. Synopsys's Design Compiler has the ability to automatically insert scan chains but hard instantiation of scan flip-flops (or inference of scan with << and >> operators) will prove simpler for your project. Finally, keep in mind that D flip-flops and scan flip-flops are expensive in terms of area. You should be able to clearly identify the

need for any flip-flops in your controller; then make sure that you don't accidentally infer more flip-flops by checking our synthesis result.

Procedure

Modeling

Describe your controller's functionality using Verilog hardware description language. Simulate this Verilog description of your controller along with your datapath, PC and memories prior to synthesis to ensure correct operation. Partitioning your controller into smaller modules can be helpful in monitoring the quality of the synthesized result, particularly if you're new to synthesis.

Synthesizing and Compiling

Having tested the Verilog model, you can then synthesize it in Synopsys's Design Compiler. Synopsys will take the Verilog description along with timing/area constraints (provided by you) and attempt to generate a gate-level netlist that meets these constraints. You should set the timing constraints based on your knowledge of the timing of your datapath and any estimates you may have for associated peripheral circuits (including memories). If you do choose to partition your design, you should set constraints only at the highest level. Once you've synthesized a gate-level netlist, you have the option of checking correct functionality prior to APR by simulating in NCVerilog. Since APR is relatively simple for controllers of this size, you can skip this step and verify functionality with better delay information after completing APR.

APR with Cadence's SOC-Encounter should be relatively simple. With your chip-level floorplan in mind, choose a good aspect ratio for your controller and distribute pins along the periphery in such a way to ease routing at the chip-level when you route the controller, datapath, and other large blocks. Once you've completed APR (and all functional verification), export your layout to Virtuoso and run DRC and LVS. The schematic you use for NCVerilog should contain the controller, datapath, and memories. You should not, however, create a layout yet to match this schematic. This will be done as part of your final project, top-level integration. You can use the on-chip memory models in the class directory for off-chip memory as well if you are unable to find Verilog descriptions of the off-chip memory you've selected. In this case, you'll need to modify size and delay parameters for the memory model to better mimic the external memory you've selected. Export SDF from SOC-Encounter and verify that the synthesized controller works with your datapath.

We provide a simple test file for you if you want to use the whole processor to test your controller now. The IMEM (ROVSD16x1024) use a testfile in the resource/codefile/ folder and the reference assemble code is in ibm13/assembler/testfile.asm. You can also use ibm13/assembler/427asm to convert your own assemble code to a binary format.

Comments & FAQ

- Plan your team's time in advance. It might take a while before you get used to Verilog syntax. Debugging could be time-consuming, since Verilog does not give any details about your bugs.
- Q: Help! NCVerilog shows Xs when I try to run my synthesized netlist.
  A: Check your synthesis log and see if there are any Warnings/Errors after the command "check_design" was run. Although these issues are innocuous in Synthesis, there may be tool incompatibilities between NCVerilog and Design Compiler that cause NCVerilog to fail. Read through this list of issues and fix them one by one. Rerun synthesis.

Submission

Please refer to the submission guidelines mentioned in the CAD1 assignment, as they apply to all CAD assignments. Do not lose points over trivial mistakes like incorrect file names, etc!

For CAD8, you need to submit the following:

- Controller:
  - Behavioral Verilog for the Controller (make your code clean and easy to understand)
  - Design Compiler scripts for the Controller
  - Encounter APR scripts for the Controller
  - Verilog simulation waveforms for both behavioral Verilog and post-APR netlist (*.apr.v + *.apr.sdf). You need to show waveforms for all the instructions. You may have additional instructions, interrupts or special operating modes that are not part of the basic requirements. While you should design your controller to support any of these extras, you do not need to demonstrate their proper operation for this assignment. You will have to show proper operation of these for your final project demo, but not for CAD8.
  - Layouts/Schematic for the Controller.
  - Clean DRC and LVS report.
- README:
  - Worst case delay number
  - Worst case delay path
  - Describe how you decide your timing constraint
  - Describe how you decide your floorplan ratio
  - Describe your controller design. Discuss any extra instructions, features, etc. that are not part of the baseline and any other design considerations you took into account.
  - Full paths to your designs.

Deadline

You must complete CAD8 by **MAr. 17, 2022** by **11pm**.
Do not modify any files in your CAD directory after that time.